Chap. 7 The Thumb Instruction Set

7.1 The Thumb bit in the CPSR

- Thumb instruction set
  - Address the issue of code density
  - May be viewed as a compressed form of a subset of the ARM instruction set
  - Implementations of Thumb use dynamic decompression in an ARM instruction pipeline and then instructions as standard ARM instructions within the processor
  - Is not a complete architecture
- The Thumb bit in the CPSR
  - ARM processors which support the Thumb instruction set can also execute the standard 32-bit ARM instruction set (bit 5 of the CPSR, the T bit)
    - T is set ⇒ processor interprets the instruction stream as 16-bit Thumb instructions
    - Otherwise ⇒ standard ARM instructions

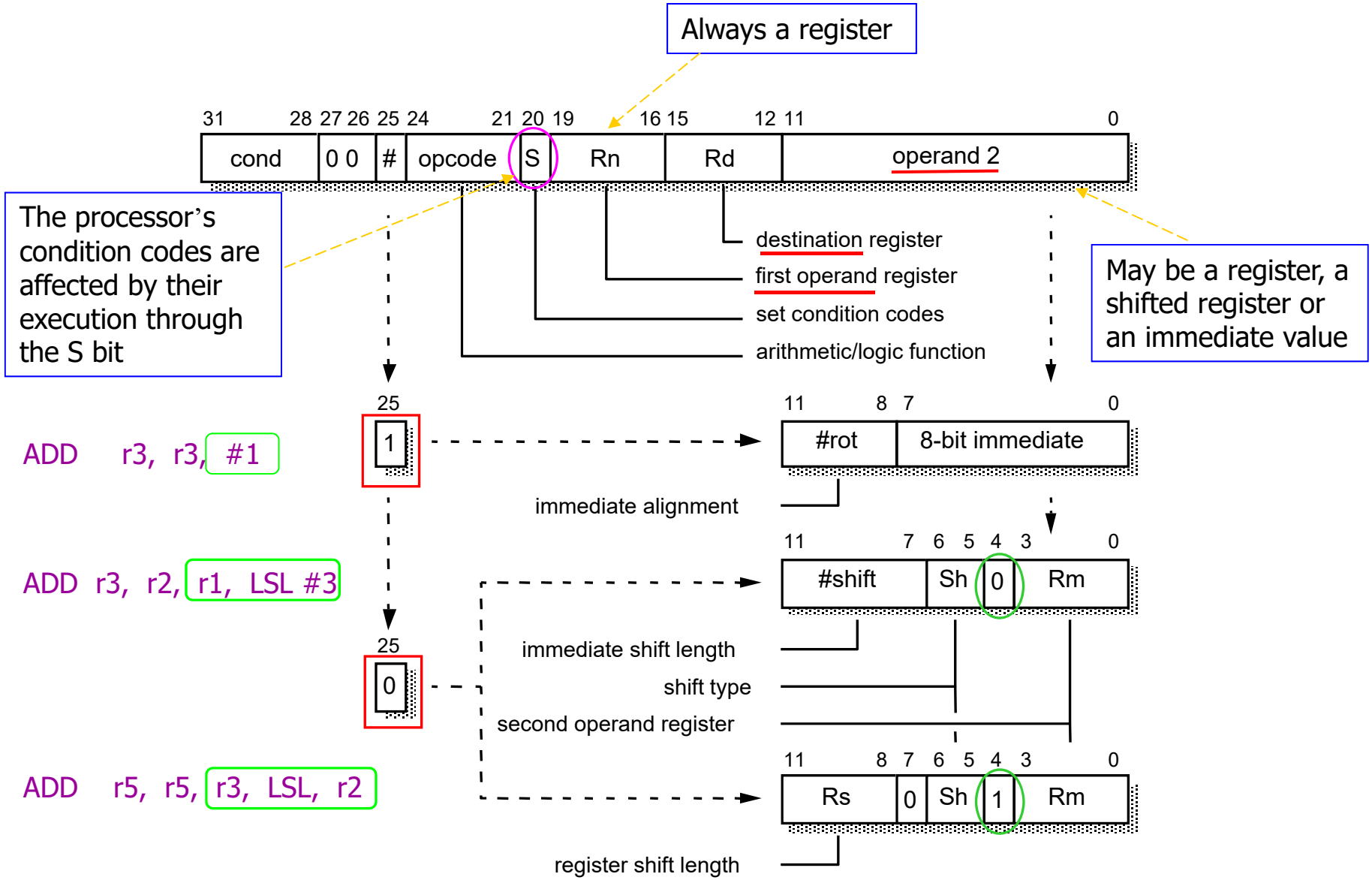


Fig. 5.6 Data processing instruction binary encoding

Comparison Code

- ARM code
  - r0: value, r1: divisor, r2: modulus, r3: DIVide

```
        MOV     r3, #0
Loop
        SUBS    r0, r0, r1
        ADDGE   r3, r3, #1
        BGE     loop
        ADD     r2, r0, r1
```

- 5x4 =20 bytes

- Thumb code
  - r0: value, r3: DIVide

```
        MOV     r3, #0
Loop
        ADD     r3, #1
        SUB     r0, r1
        BGE     loop
        SUB     r3, #1
        ADD     r2, r0, r1
```

- 6x2 =12 bytes

Thumb entry and exit

- ARM cores start up, after reset, executing ARM instruction.
- Thumb entry
  - Execution of a Branch and Exchange instruction (BX)
  - Exception return if the interrupted instruction belongs to Thumb mode.
- Thumb exit
  - An explicit switch back to ARM.
  - The occurrence of Exception since exception entry is always handled in ARM code.

7.1 The Thumb bit in the CPSR

- Thumb entry and exit (Branch and Exchange instruction)

```
                CODE32                  ;  ARM code follows
. .
                BLX         TSUB        ; call Thumb subroutine
        . .
                CODE16                  ; start of Thumb code
TSUB    . .                             ; Thumb subroutine
                BX          r14         ; return to ARM code
```

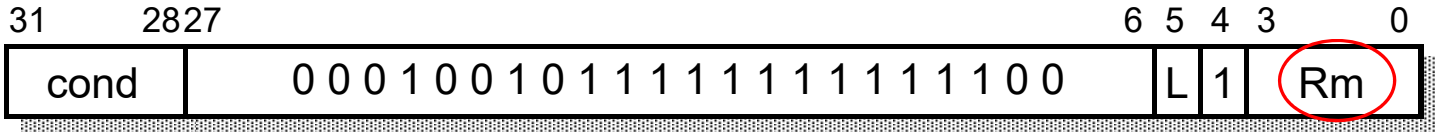
ARM CPSR format

5.5 Branch, Branch with Link and eXchange (BX, BLX)

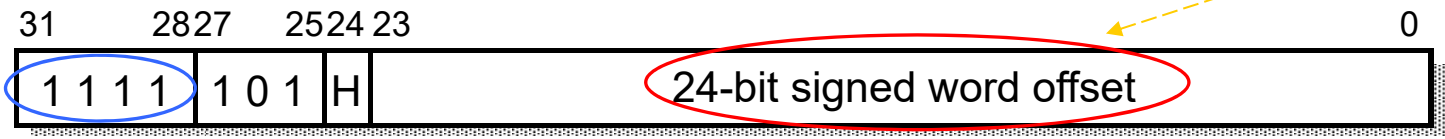
- BX, BLX
  - These instructions are available on ARM chips which support the Thumb (16-bit) instruction set, and are a mechanism for switching the processor to execute Thumb instructions or for returning symmetrically to ARM and Thumb calling routines

- Binary encoding

(1) `BX|BLX Rm`



(2) `BLX label`



branch target

Fig. 5.4 Branch (with optional link) and exchange instruction binary encoding

5.5 Branch, Branch with Link and eXchange (BX, BLX)

- Description (Cont.)
  - In the *first format* the branch target is specified in a register, Rm
  - Bit[0] of Rm is copied into the **T** bit in the CPSR and bits[31:1] are moved into the PC:

    - If Rm[0] is 1, the processor switches to execute Thumb instructions and begins executing at the address in Rm aligned to a half-word boundary by clearing the bottom bit (…..0)
    - If Rm[0] is 0, the processor continues executing ARM instructions and begins executing at the address in Rm aligned to a word boundary by clearing Rm[1] (….00)
  - In the *second format* the branch target is an address computed by sign extending the 24-bit offset specified in the instruction
  - Format (1) instructions may be executed conditionally or unconditionally, but format (2) instructions are executed unconditionally

5.5 Branch, Branch with Link and eXchange (BX, BLX)

- Assembler format :

  1: B{L}X{<cond>} Rm

  2: BLX <target address>

  <target address> is normally a label

- Example

```
                CODE32                  ;  ARM code follows
. .
                BLX         TSUB        ; call Thumb subroutine
          . .
                CODE16                   ; start of Thumb code
TSUB    . .                              ; Thumb subroutine
                BX          r14          ; return to ARM code
```

Return from BL subroutine call

- MOV pc, lr
- BX lr
- POP {pc}

example

- ;ARM code

```
    CODE32
    LDR     r0,=thumbCode+1
    MOV lr, pc
    BX      r0
```

- Thumb code

```
    CODE16
thumbCode
    ADD     r1, #1
    BX      lr
```

- ;ARM code

```
    CODE32
    LDR     r0,=thumbCode+1
    BLX     r0
```

- Thumb code

```
    CODE16
thumbCode
    ADD     r1, #1
    BX      lr
```

7.2 The Thumb programmer's model

- Thumb programmer's model
  - Thumb instruction set is a subset of the ARM instruction set and the instructions operate on a restricted view of the ARM registers
  - The programmer's model is illustrated in Fig. 7.1
    - gives full access to the eight 'Lo' general purpose registers r0 to r7
    - makes extensive use of r13 to r15 for special purposes
      - r13: stack pointer
      - r14: link register
      - r15: program counter (PC)
    - the remaining registers (r8 to r12 and the CPSR) have only restricted access
      - a few instructions allow the 'Hi' registers (r8 to r15) to be specified
      - The CPSR condition code flags are set by arithmetic and logical operations and control conditional branching

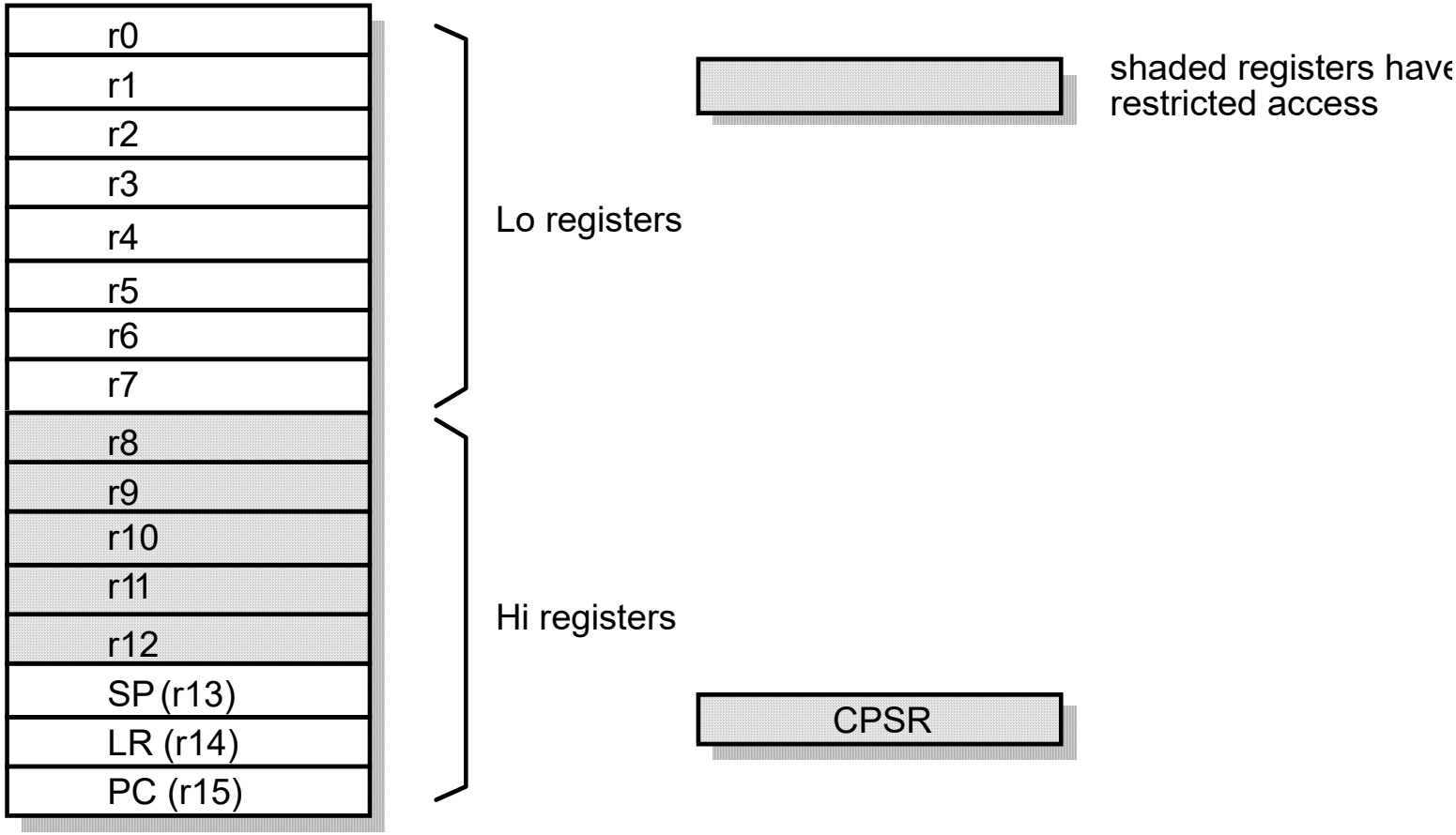


Fig. 7.1 Thumb accessible registers

7.2 The Thumb programmer's model

- Thumb-ARM similarities
  - Thumb instructions map onto ARM instructions so they inherit many properites of the ARM instruction set
    - The load-store architecture with data processing, data transfer and control flow instructions
    - Support for 8-bit byte, 16-bit half-word and 32-bit word data types where half-words are aligned on 2-byte boundaries and words are aligned on 4-byte boundaries
    - A 32-bit unsegmented memory
- Thumb-ARM differences
  - In order to achieved 16-bit instruction long
    - Most Thumb instructions are executed unconditionally
    - Many Thumb data processing instructions use a 2-address format (destination register is the same as one of the source registers)
    - Thumb instruction formats are less regular than ARM instruction formats

7.2 The Thumb programmer's model

- Thumb exceptions
  - All exceptions return the processor to ARM execution and are handled within the ARM programmer's model
  - Thumb instructions are two bytes rather than four bytes long
    - The Thumb architecture requires that the link register value be automatically adjusted to match the ARM return offset
    - Allow the same return instruction to work in both cases, rather than have the return sequence made more complex

The case where the return address is in r14 :

- To return from a SWI or undefined instruction trap use

```
MOVS    pc,  r14
```

- To return from an IRQ, FIQ or prefetch abort use

```
SUBS    pc,  r14,  #4
```

- To return from a data abort to retry the data access use

```
SUBS    pc,  r14,  #8
```

7.3 Thumb branch instruction

- Control flow instructions
  - PC-relative branch instruction, branch-and-link instruction, and branch-and-exchange instruction
  - ARM instructions have a large (24-bit) offset field
    - which clearly will not fit in a 16-bit instruction format
- Binary encodings (Fig. 7.2)
- Description
  - Typical uses of branch instructions include
    - short conditional branches to control (for example) loop exit
    - medium range unconditional branches to ‘goto’ sections of code
    - long-range subroutine calls

5.4 Branch and Branch with Link (B, BL)

- Binary encoding

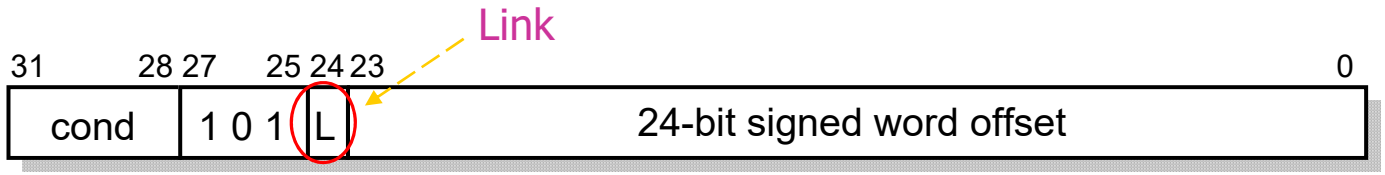


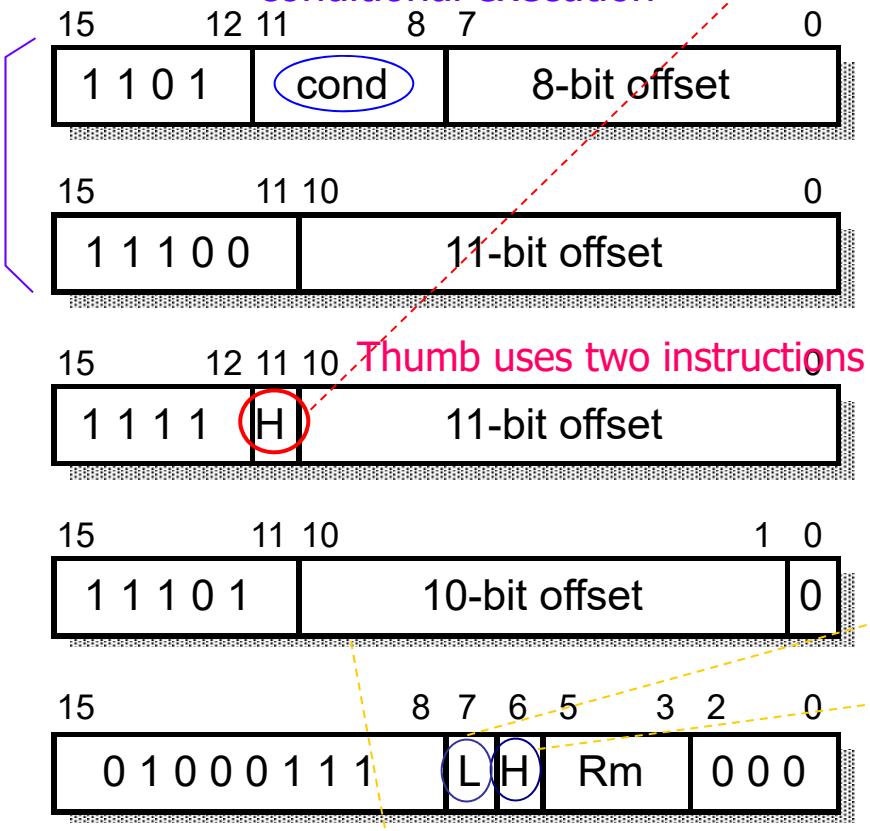
Fig. 5.3 Branch and Branch with Link binary encoding

- Description
  - Branch and branch with Link instructions cause the processor to begin executing instructions from an address computed by sign extending the 24-bit offset specified in the instruction
  - The Branch with Link variant, which has the L bit (bit 24) set, also moves the address of the instruction following the branch into the link register (r14) of the current processor mode

1. H=0 LR:=PC+(sign-extended offset shifted left 12 places);
2. H=1 PC:=LR+(offset shifted left 1 place); LR:=oldPC+3. (Chap.5, p.20)

2+1: caller is a Thumb routine

conditional execution



(1) `B<cond> <label>`

(2) `B <label>`

Thumb uses two instructions to give a combined 22-bit half-word offset

(3) `BL <label>`

Link ?

(3a) `BLX <label>`

'H' can be set to select a 'Hi' register (r8 to r15)

(4) `B{L}X Rm`

An alternative second step which gives the BLX variant
1. (BL, H=0) LR:=PC+(sign-extended offset shifted left 12 places);
2. (BLX) PC:=LR+(offset shifted left 1 place)&0xfffffffc; LR:=oldPC+3;

Target is a ARM instruction

the Thumb bit is cleared

2+1: caller is a Thumb routine

BL example

- For the following case:

```
pc=1024                BL target
pc=1028                MOV r1,r2
pc=1030                SUB r3,r1
                       ..............
pc=5146      target    ADD r1, r2
                       ..............
```

BL instruction will be translated as follows:

- PC Offset: 4122 => Instruction (half-word) offset: 2061
- 2061 = 0b 00000000001 00000001011
- For pc=1024=> lr= pc + 00000000001 <<12 = 5120
- For pc =1026=> pc=lr + 00000001011<<1 = 5146

lr = pc + 3 = 1029

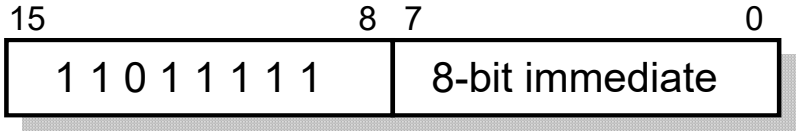
pc=1024	1111 1 00000000001
pc=1026	1111 0 00000001011

7.3 Thumb branch instruction

- Equivalent ARM instruction
  - The formats 1 to 3 of Thumb branch instructions are very similar to the ARM branch instructions
  - ARM instructions support only word (4-byte) offsets whereas the Thumb instructions require half-word (2-byte) offsets
    - There is no direct mapping from these Thumb instructions into the ARM instruction set
    - The ARM cores that support Thumb are slightly modified to support half-word branch offsets, with ARM branch instructions being mapped to even half-word offsets
  - Format 4 is equivalent to the ARM instruction with the same assembler syntax

7.4 Thumb software interrupt instruction

- Thumb software interrupt instruction
  - behaves exactly like the ARM equivalent and the exception entry sequence causes the processor to switch to ARM execution
- Binary encoding



- Description
  - The instruction causes the following actions
    - The address of the next Thumb instruction is saved in r14_svc
    - The CPSR is saved in SPSR_svc
    - The processor disables IRQ, clears the Thumb bit and enters supervisor mode by modifying the relevant bits in the CPSR
    - The PC is forced to address 0x08 (Chap. 5, p.11)
  - The ARM instruction SWI handler is then entered

7.4 Thumb software interrupt instruction

- Equivalent ARM instruction
  - The Equivalent ARM instruction has an identical assembler syntax
  - The 8-bit immediate is zero-extended to fill the 24-bit field in the ARM instruction
  - This limits the SWIs available to Thumb code to the first 256 of the 16 million potential ARM SWIs

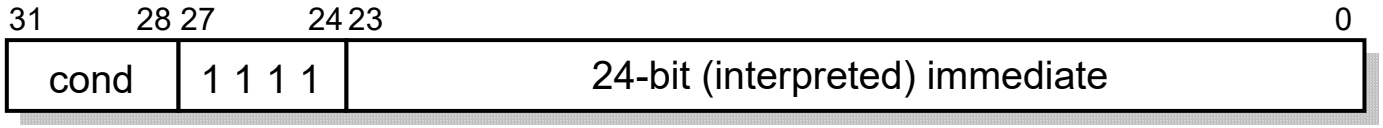


Fig. 5.5 Software interrupt binary encoding

7.5 Thumb data processing instructions

- Thumb data processing instructions
  - comprise a highly optimized set of fairly complex formats covering the operations most commonly required by a complier
  - The functions of these instructions are clear enough
  - The selection: based on a detailed understanding of the needs of typical application programs
- Binary encodings (Fig. 7.4)
- Description
  - These instructions all map onto ARM data processing instructions
  - The Thumb instruction set separates shift and ALU operations into separate instructions
    - the shift operation is presented as an opcode rather than an operand modifier

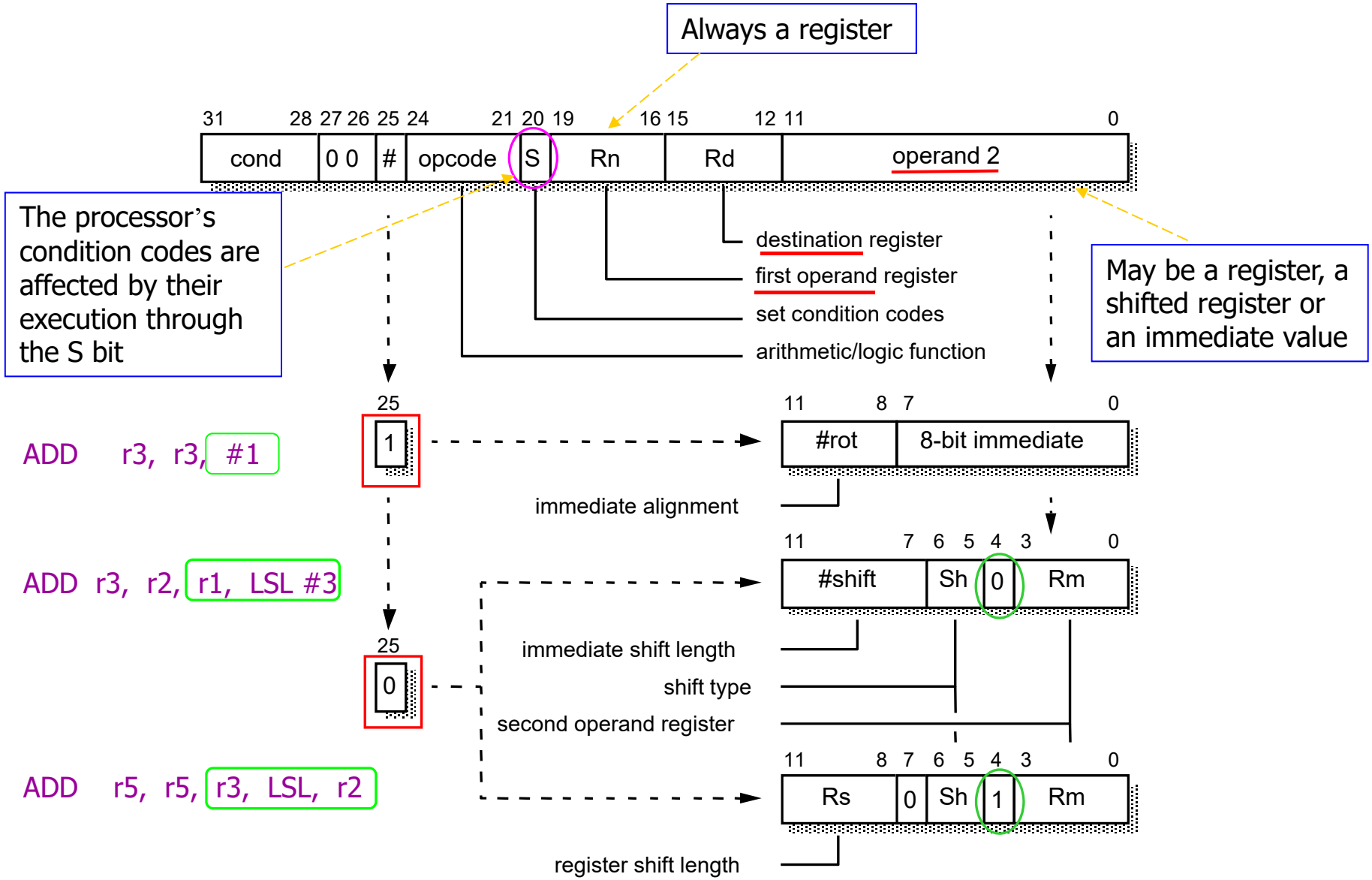


Fig. 5.6 Data processing instruction binary encoding

7.5 Thumb data processing instructions

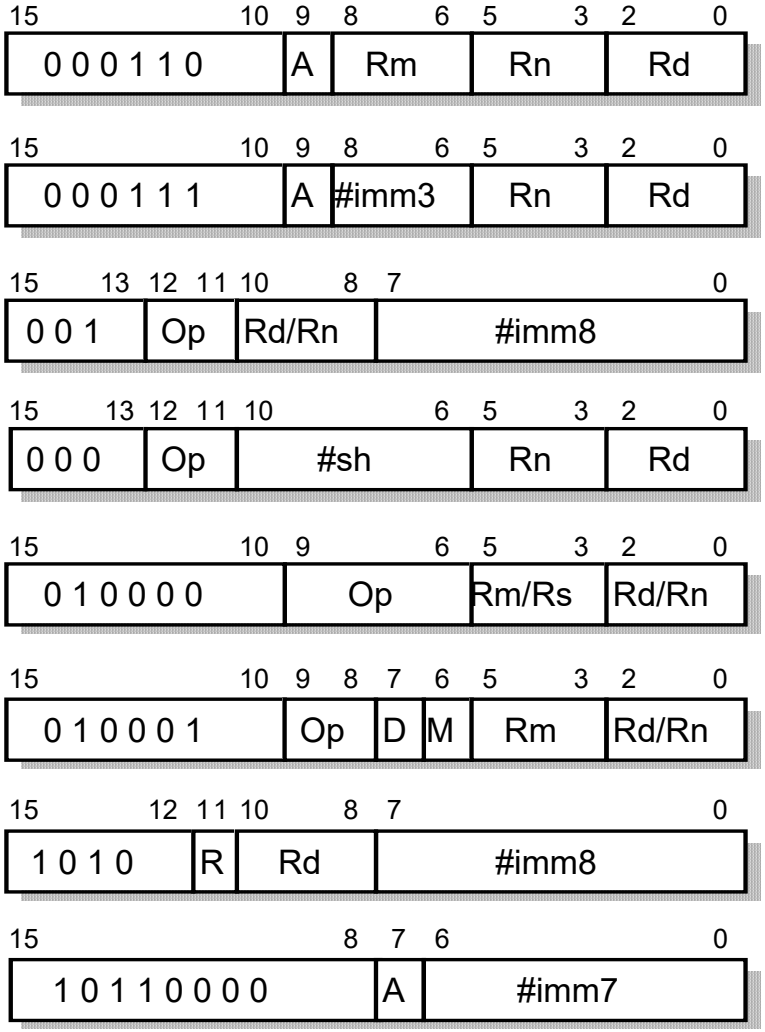


Fig. 7.4 Thumb data processing instruction binary encodings

7.5 Thumb data processing instructions

- Equivalent ARM instructions
  - Instructions that use the 'Lo', general-purpose registers (r0 to r7)

; ARM instruction		Thumb instruction	
MOVS	Rd, #<#imm8>	; MOV	Rd, #<#imm8>
MVNS	Rd, Rm	; MVN	Rd, Rm
CMP	Rn, #<#imm8>	; CMP	Rn, #<#imm8>
CMP	Rn, Rm	; CMP	Rn, Rm
CMN	Rn, Rm	; CMN	Rn, Rm
TST	Rn, Rm	; TST	Rn, Rm
ADDS	Rd, Rn, #<#imm3>	; ADD	Rd, Rn, #<#imm3>
ADDS	Rd, Rd, #<#imm8>	; ADD	Rd, #<#imm8>
ADDS	Rd, Rn, Rm	; ADD	Rd, Rn, Rm
ADCS	Rd, Rd, Rm	; ADC	Rd, Rm
SUBS	Rd, Rn, #<#imm3>	; SUB	Rd, Rn, #<#imm3>
SUBS	Rd, Rd, #<#imm8>	; SUB	Rd, #<#imm8>
SUBS	Rd, Rn, Rm	; SUB	Rd, Rn, Rm
SBCS	Rd, Rd, Rm	; SBC	Rd, Rm
RSBS	Rd, Rn, #0	; NEG	Rd, Rn,
MOVS	Rd, Rm, LSL #<#sh>	; LSL	Rd, Rm, #<#sh>

7.5 Thumb data processing instructions

- Equivalent ARM instructions
  - Instructions that use the 'Lo', general-purpose registers (r0 to r7)

```
;  ARM instruction                    Thumb instruction
   MOVS    Rd, Rd,  LSL Rs        ;  LSL     Rd, Rs
   MOVS    Rd, Rm, LSR #<#sh>     ;  LSR     Rd, Rm, #<#sh>
   MOVS    Rd, Rd,  LSR Rs        ;  LSR     Rd, Rs
   MOVS    Rd, Rm, ASR #<#sh>     ;  ASR     Rd, Rm, #<#sh>
   MOVS    Rd, Rd,  ASR Rs        ;  ASR     Rd, Rs
   MOVS    Rd, Rd,  ROR Rs        ;  ROR     Rd, Rs
   ANDS    Rd, Rd,  Rm            ;  AND     Rd, Rm
   EORS    Rd, Rd,  Rm            ;  EOR     Rd, Rm
   ORRS    Rd, Rd,  Rm            ;  ORR     Rd, Rm
   BICS    Rd, Rd,  Rm            ;  BIC     Rd, Rm
   MULS    Rd, Rm, Rd             ;  MUL     Rd, Rm
```

7.5 Thumb data processing instructions

- Equivalent ARM instructions
  - Instructions that operate with or on the the 'Hi' registers (r8 to r15), in some cases in combination with a 'Lo' register

```
;  ARM instruction                    Thumb instruction
   ADD     Rd, Rd,  Rm                ;  ADD     Rd, Rm  (1/2 Hi regs)
   CMP     Rn, Rm                     ;  CMP     Rn, Rm  (1/2 Hi regs)
   MOV     Rd, Rm                     ;  MOV     Rd, Rm  (1/2 Hi regs)
   ADD     Rd, PC, #<#imm8>           ;  ADD     Rd, PC, #<#imm8>
   ADD     Rd, SP, #<#imm8>           ;  ADD     Rd, SP, #<#imm8>
   ADD     SP, SP, #<#imm7>           ;  ADD     SP, SP, #<#imm7>
   SUB     SP, SP, #<#imm7>           ;  SUB     SP, SP, #<#imm7>
```

7.5 Thumb data processing instructions

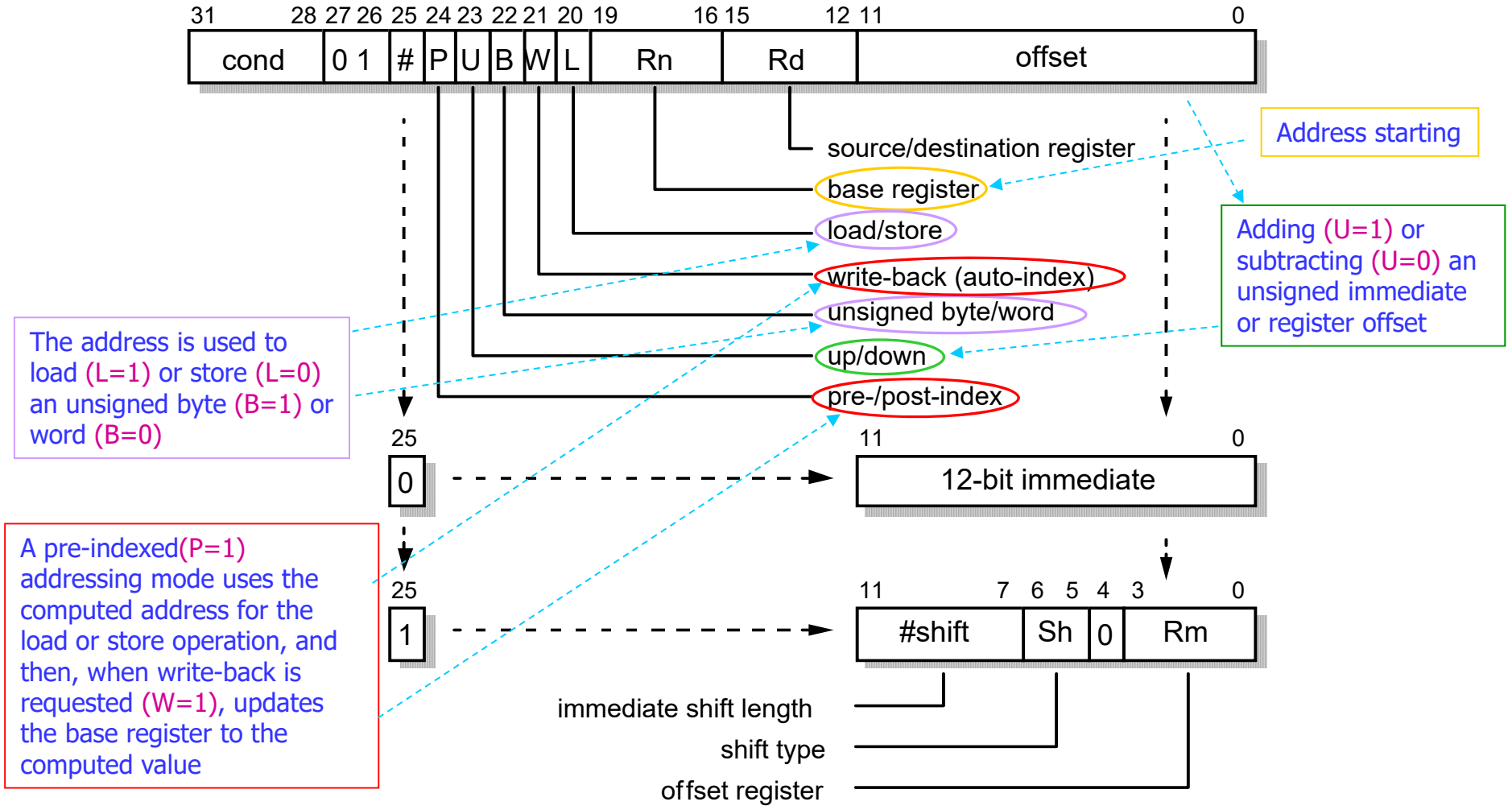
- Notes
  - *All* the data processing instructions that operate with and on the 'Lo' registers update the condition code bits (S bit in the equivalent ARM instruction)
  - The instructions that operate with and on the 'Hi' registers do not changes the condition code bits, with the exception of CMP
  - The instructions that are indicated above as requiring '1 or 2 Hi regs' must have one or both register operands specified in the 'Hi' register area

Some instruction execution example

- Example1
  - PRE
    - cpsr=nzcvIFT_SVC
    - r1 = 0x80000000
    - r2 = 0x10000000
    - ADD r0, r1, r2
  - POST
    - cpsr=NzcvIFt_SVC
    - r1 = 0x90000000
- Example2
  - PRE
    - r2 = 0x00000002
    - r4 = 0x00000001
    - LSL r2, r4
  - POST
    - r2 = 0x00000004
    - r4 = 0x00000001
- Example3
  - PRE
    - Mem32[0x90000]=0x00000001
    - Mem32[0x90004]=0x00000002
    - Mem32[0x90008]=0x00000003
    - r0 = 0x00000000
    - r1 = 0x00090000
    - r4 = 0x00000004
    - LDR r0, [r1, r4]
  - POST
    - r0 = 0x00000002
    - r1 = 0x00090000
    - r4 = 0x00000004
    - LDR r0, [r1, #0x4]
  - POST
    - r0 = 0x00000002

7.6 Thumb single register data transfer instructions

- Thumb single register data transfer instructions
  - Choice: based on the sort of things that compilers like to do frequently
  - Larger offsets for accesses to the literal pool (PC-relative) and to the stack (SP-relative)
  - Restricted support given to signed operands (base plus register addressing only) compared with unsigned operands (base plus offset or register)
    - As with the ARM instructions, the signed variants are only supported by the load instructions since store signed and unsigned have exactly the same effect
- Description
  - These instructions have exactly the same semantics as the ARM equivalent (have identical assembler formats)
  - In all cases the offset is scaled to the size of the data type, so, for instance, the range of the 5-bit offset is
    - 32 bytes in a load or store byte instruction
    - 64 bytes in a load or store half-word instruction
    - 128 bytes in a load or store word instruction

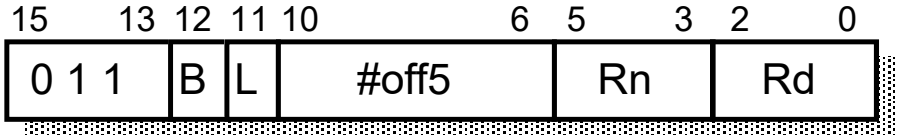


LDRLS pc, [r1, r0, LSL #2] Chap.3, p.35

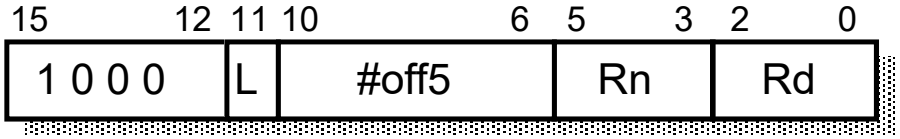
Fig 5.9 Single word and unsigned byte data transfer instruction binary encoding

7.6 Thumb single register data transfer instructions

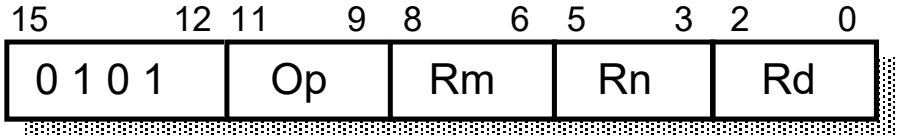
- Binary encodings



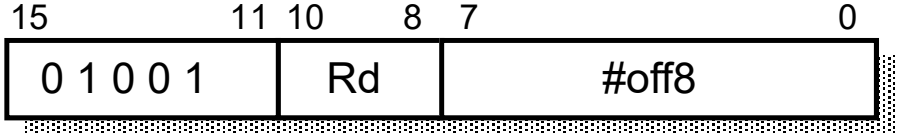
(1) `LDR|STR{B} Rd,[Rn,#off5]`



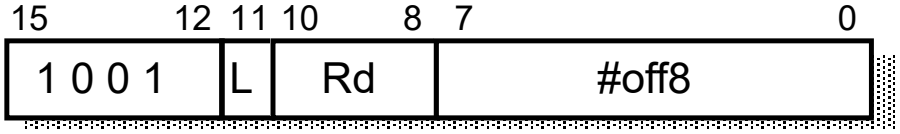
(2) `LDRH|STRH Rd,[Rn,#off5]`



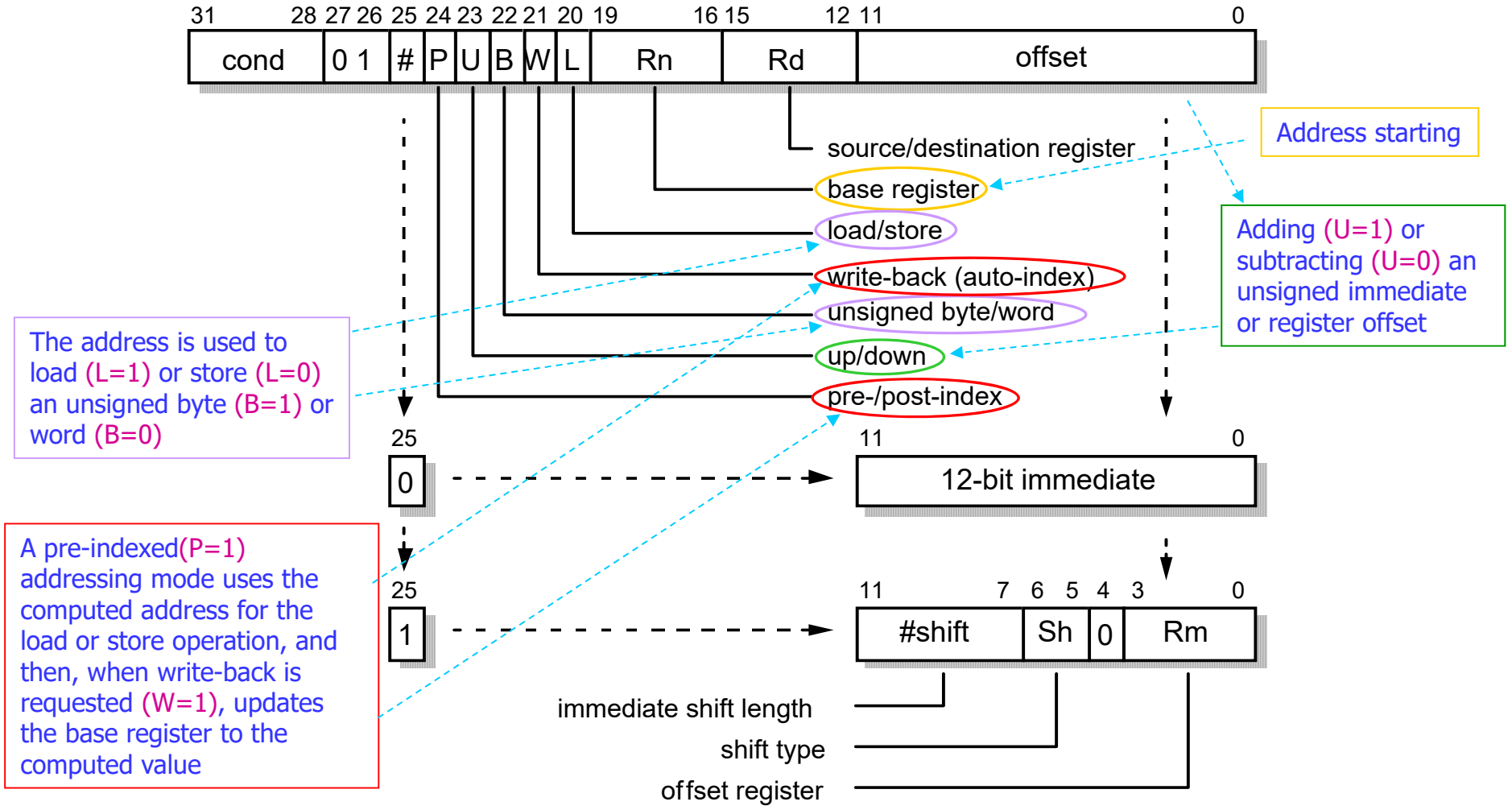
(3) `LDR|STR{S}{H|B} Rd,[Rn,Rm]`



(4) `LDR Rd,[PC,#off8]`



(5) `LDR|STR Rd,[SP,#off8]`



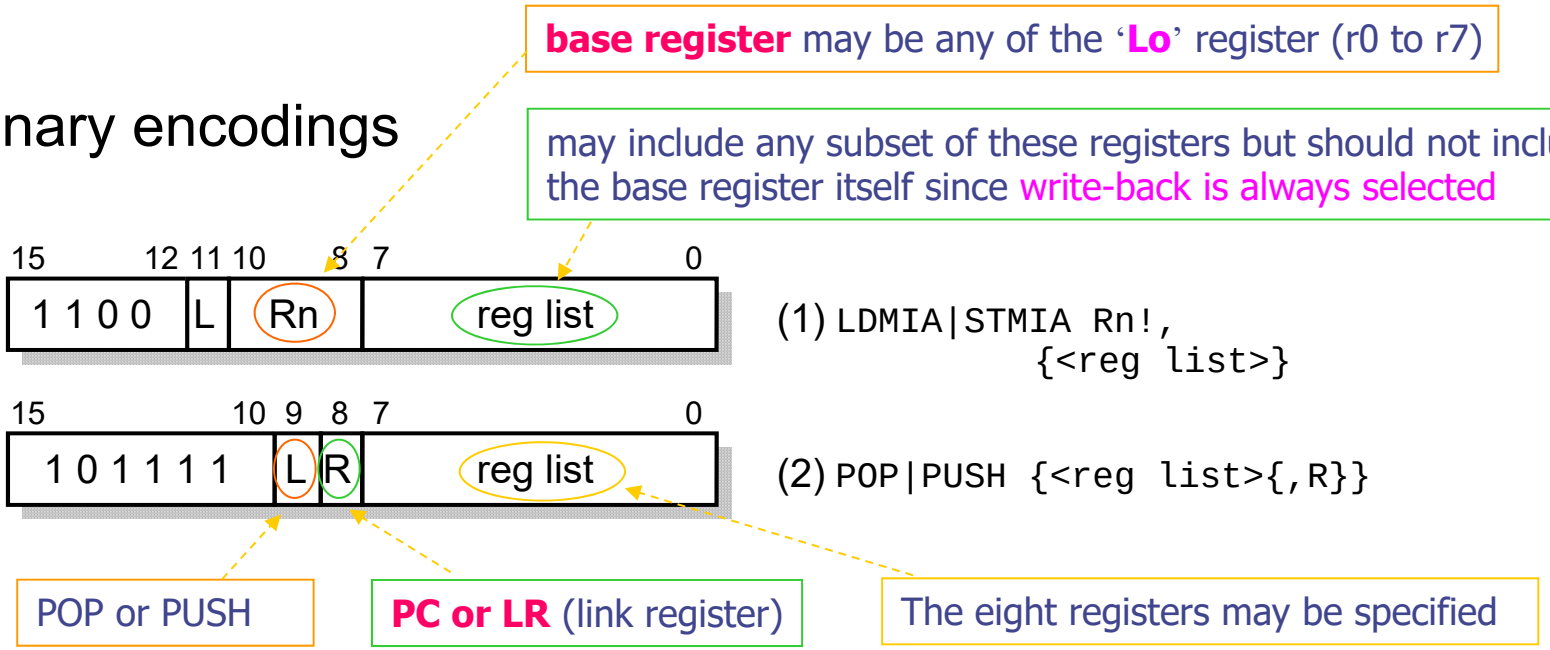
LDRLS pc, [r1, r0, LSL #2] Chap.3, p.35

Fig 5.9 Single word and unsigned byte data transfer instruction binary encoding

7.7 Thumb multiple register data transfer instructions

- Thumb multiple register data transfer instructions
  - As in the ARM instruction set, these instruction are useful both for procedure entry and return and for memory block copy
  - The tighter encoding means
    - The two uses must be separated and the number of addressing modes restricted

- Binary encodings



7.7 Thumb multiple register data transfer instructions

- Description
  - The stack forms use SP (r13) as the base register and always use write-back
  - The stack model is fixed as full-descending
  - link register (LR, or r14) may be included in the 'PUSH' instruction
  - PC (r15) may be included in the 'POP' form
- Equivalent ARM instruction
  - The equivalent ARM instruction have the same assembler format in the first two cases
  - Replace POP and PUSH with the appropriate addressing mode in the second two cases

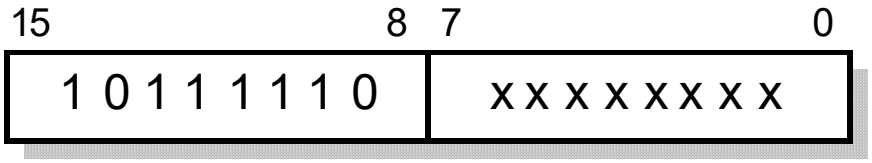
```
Block copy:   LDMIA  Rn!,  {<reg list>}
              STMIA  Rn!,  {<reg list>}

Pop:          LDMFD  SP!,  {<reg list>{,  pc}}
Push:         STMFD  SP!,  {<reg list>{,  lr}}
```

7.8 Thumb breakpoint instruction

- Thumb breakpoint instruction
  - Behaves exactly like the ARM equivalent
  - Are used for software debugging purposes
    - cause the processor to break from normal instruction execution and enter appropriate debugging procedures

- Binary encoding



- Assembler format
  - BKPT

7.9 Thumb implementation

- Thumb implementation
  - Can be incorporated into a 3-stage pipeline ARM processor macrocell with relatively minor changes to most of the processor logic
    - The 5-stage pipeline implementations are trickier
  - The biggest addition is the Thumb instruction decompressor in the instruction pipeline
    - This logic translates a Thumb instruction into its equivalent ARM instruction (Fig. 7.8)
  - The addition of the decompressor logic might be expected to increase the decode latency
    - In fact the ARM7 pipeline does relatively little work in phase 1 of the decode cycle
    - The decompression logic can be accommodated here without compromising the cycle time or increasing the pipeline latency
    - The ARM7TDMI Thumb pipeline operates in exactly the way described in ‘The 3-stage pipeline’

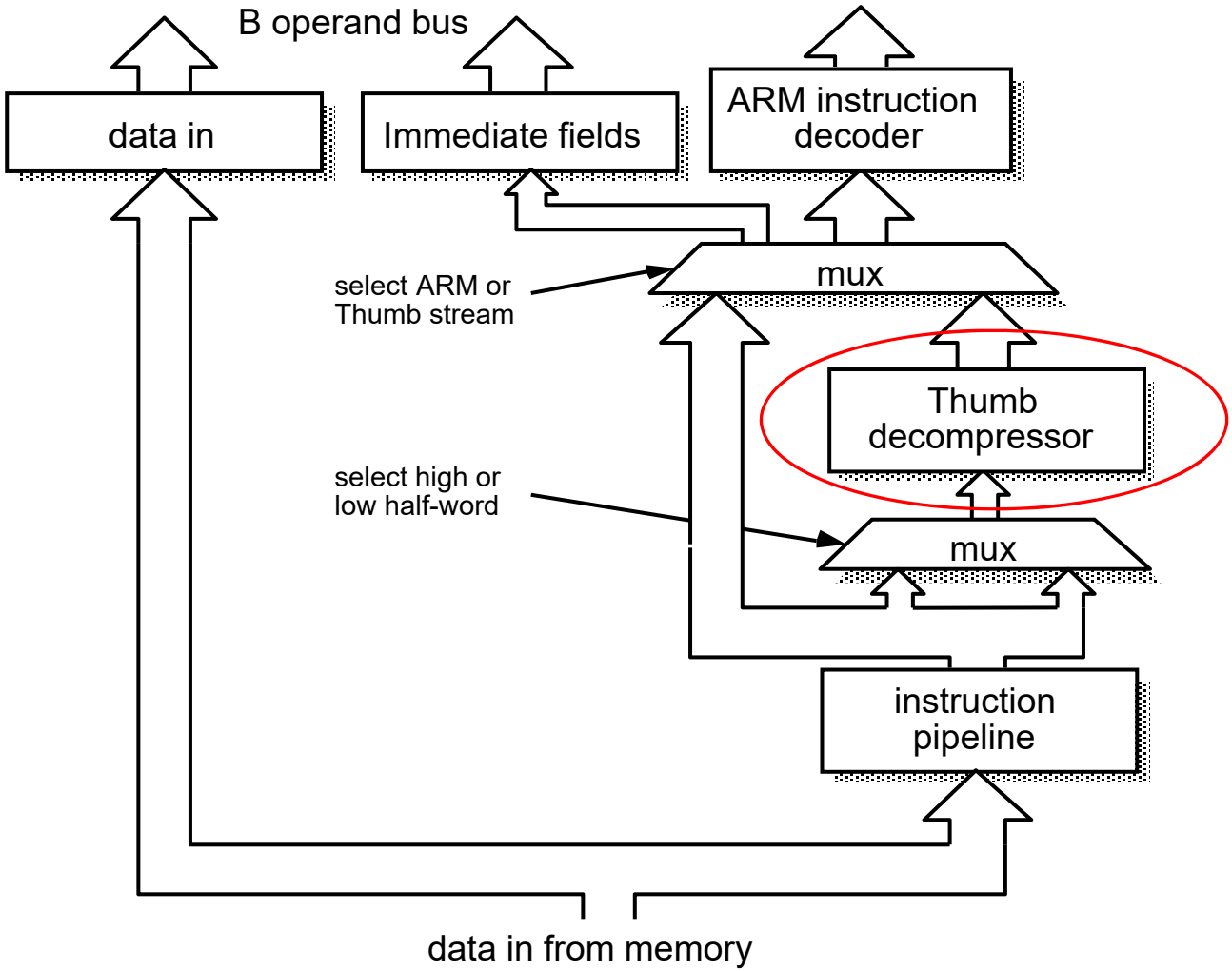


Fig. 7.8 The Thumb instruction decompressor organization

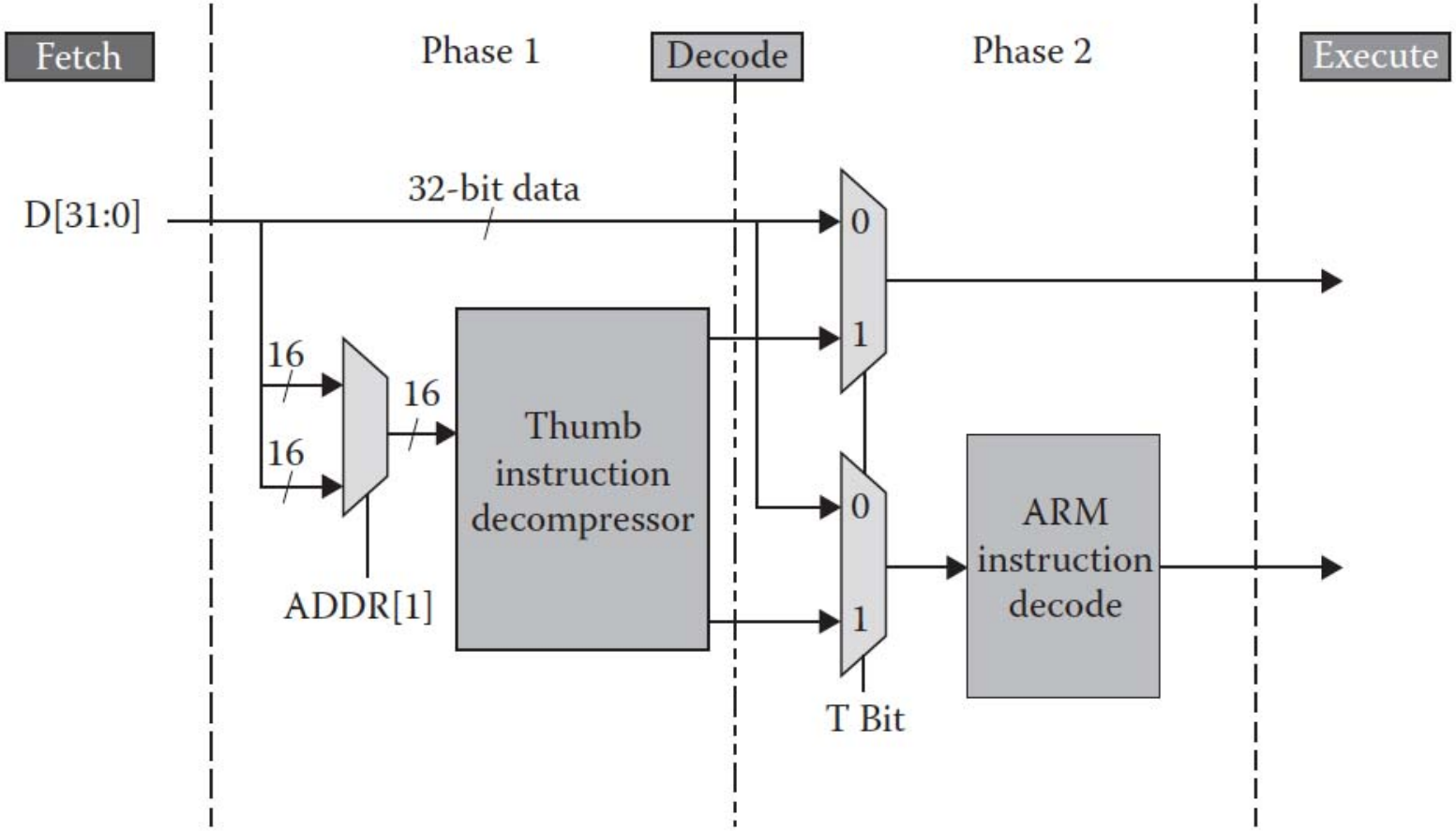


FIGURE 17.4 The ARM7TDMI processor pipeline.

7.9 Thumb implementation

- Instruction mapping
  - The Thumb decompressor performs a static translation from the 16-bit Thumb instruction into the equivalent 32-bit ARM instruction
    - Perform a look-up to translate the major or minor opcodes
    - Zero-extending the 3-bit register specifiers to give 4-bit specifiers
    - Map other fields across as required
  - An example
    - Fig. 7.9:

      Thumb ‘ADD Rd, #imm8’ $\Rightarrow$ ARM ‘ADDS Rd, Rd, #imm8’
  - Note that
    - Since the only conditional Thumb instructions are branches, the condition ‘always’ is used in translating all other Thumb instructions
    - Whether or not a Thumb data processing instruction should modify the condition codes in the CPSR is implicit in the Thumb opcode
      - this must be made explicit in the ARM instruction
    - The Thumb 2-address format can always be mapped into the ARM 3-address format

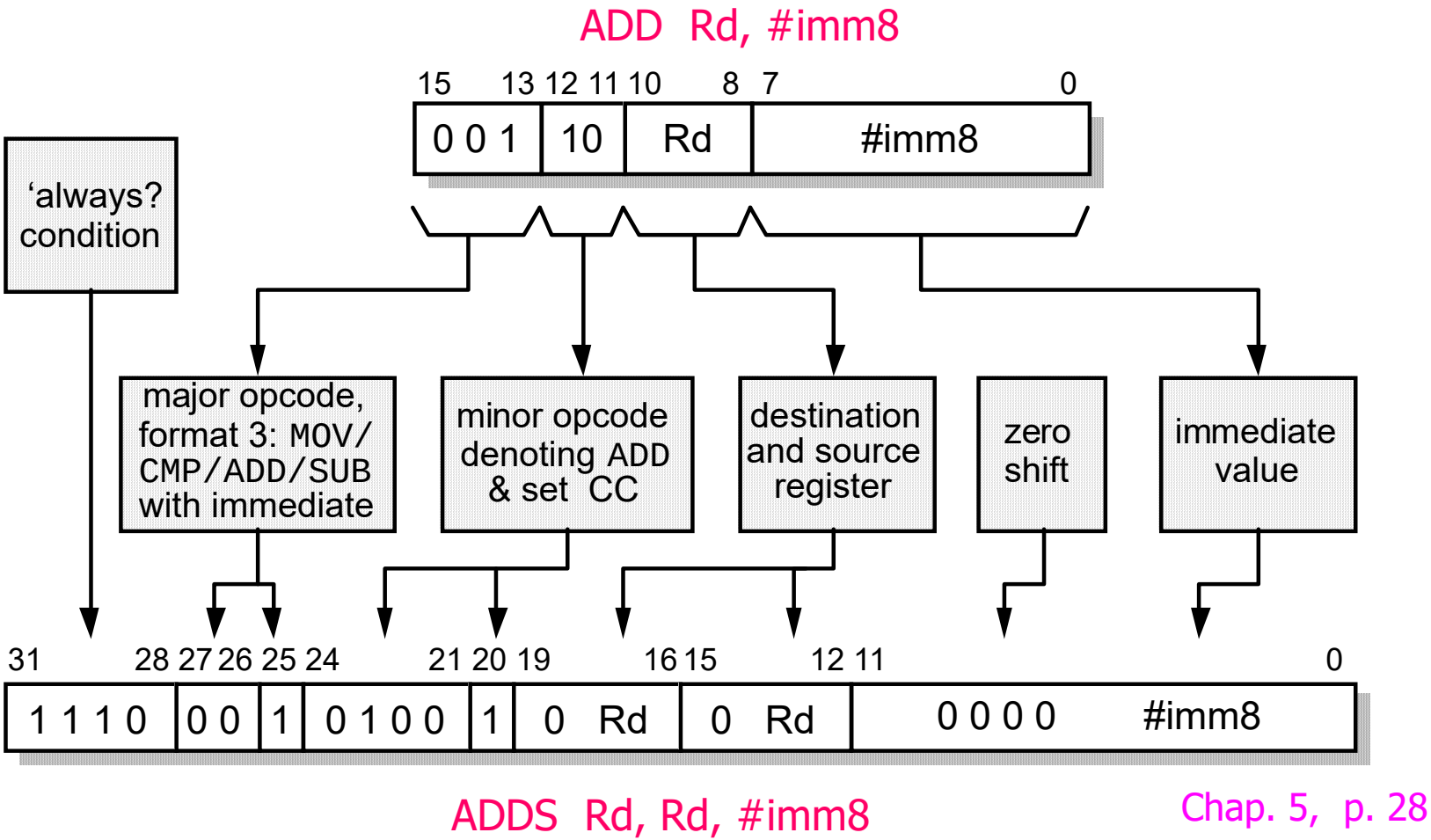


Fig. 7.9 Thumb to ARM instruction mapping

7.10 Thumb applications

- Thumb instruction
  - typically has less semantic content than an ARM instruction, a particular program will require more Thumb instructions than it would have needed ARM instructions
  - In a typical example Thumb code may require 70% of the space of ARM code
- Thumb properties
  - The Thumb code requires 70% of the space of the ARM code
  - The Thumb code uses 40% more instructions than the ARM code
  - With 32-bit memory, ARM code is 45% faster than Thumb code
  - With 16-bit memory, Thumb code is 45% faster than ARM code
  - Thumb code uses 30% less external memory power than ARM code
  - *Performance*: use 32-bit memory and run ARM code
  - *Cost and Power*: use 16-bit memory and run Thumb code

7.10 Thumb applications

- Thumb systems
  - A high-end 32-bit ARM system
    - may use Thumb code for certain non-critical routines to save power or memory requirements
  - A low-end 16-bit system
    - may have a small amount of on-chip 32-bit RAM for critical routines running ARM code, but use off-chip Thumb code for all non-critical routines
  - The second of these examples is perhaps closer to the sort of application for which Thumb was developed
    - Mobile telephone and pager applications incorporate real-time digital signal processing (DSP) functions
      - that may require the full power of the ARM, but these are tightly coded routines that can fit in a small amount of on-chip memory
    - The more complex and much larger code that controls the user interface, battery management system, and so on
      - is less time-critical, and the use of Thumb code will enable off-chip ROMs to give good performance on an 8- or 16-bit bus, saving cost and improving battery lift